JOURNAL OF SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE
*J. Softw. Maint: Res. Pract.* **11**, 19–34 (1999)

**Research**

# Simulated Annealing and Genetic Algorithms for Optimal Regression Testing

NASHAT MANSOUR* and KHALID EL-FAKIH
*Department of Computer Science, Lebanese American University, Beirut, Lebanon*

**SUMMARY**

**The optimal regression testing problem is one of determining the minimum number of test cases needed for revalidating modified software in the maintenance phase. We present two natural optimization algorithms, namely, a simulated annealing and a genetic algorithm, for solving this problem. The algorithms are based on an integer programming problem formulation and the program's control flow graph. The main advantage of these algorithms, in comparison with exact algorithms, is that they do not suffer from an exponential explosion for realistic program sizes. The experimental results, which include a comparison with previous algorithms, show that the simulated annealing and genetic algorithms find the optimal or near-optimal number of retests within a reasonable time. Copyright © 1999 John Wiley & Sons, Ltd.**

KEY WORDS: software testing; optimal retests; software revalidation; integer programming; natural optimization; Metropolis algorithm

## 1. INTRODUCTION

Maintenance is known to be an important and expensive phase in the software development life cycle (Bennett, 1991). It can be: (i) corrective, as in response to an error report, (ii) perfective when the existing system is improved or enhanced, or (iii) adaptive when the system is adapted in response to changes in its environment. For each of these cases, the software system has to be revalidated by regression testing, which is called corrective in the first case and progressive in the second and third cases (Leung and White, 1989). Regression testing aims to verify that the modifications have not caused unintended adverse side-effects and that the modified system still meets the requirements (IEEE, 1990). Regression testing is a significant component of maintenance. Hence, reducing the cost of regression testing is important for making software maintenance a less expensive activity.

For regression testing, it is costly to repeat the whole set of test cases used in the initial development of the program. Moreover, it is unreliable to choose a random subset of test cases. Therefore, it is important to select a subset of test cases that has minimal cardinality and yet

*Correspondence to: Dr. Nashat Mansour, Department of Computer Science, Lebanese American University, 475 Riverside Drive, Suite 1846, New York NY 10115–0065, USA. Email: nmansour@lau.edu.lb
Contract/grant sponsor: Lebanese National Council for Scientific Research

CCC 1040–550X/99/010019–16$17.50
Copyright © 1999 John Wiley & Sons, Ltd.
*Received 14 January 1998*
*Revised 13 November 1998*

accomplishes the goal of regression testing. This problem is henceforth referred to as optimal regression testing, or simply optimal retesting.

A number of regression testing methods have been described in the literature. Further, some software tools have been constructed (Chen, Rosenblum and Vo, 1994; White *et al.*, 1993), and a theoretical analysis framework has been recently presented for comparing regression testing algorithms (Rothermel and Harrold, 1996).

Leung and White (1989) have suggested classifying the initial test cases as reusable, retestable, obsolete, and adding new-structural and new-specification test cases. Then, corrective regression testing may involve test cases from the reusable, retestable and new-structural classes, whereas progressive regression testing may involve test cases from all five classes. Leung and White (1990) have extended this method to cover programs with global variables. Also, Leung and White (1992) have proposed building ‘firewalls’ to confine integration testing to a small set of modules rather than allowing it to spread to many other modules. The construction of a firewall links the modified modules with their direct ascendants and direct descendants. A similar concept of class firewall has been proposed for retesting objected-oriented software systems (Hsia *et al.*, 1997).

A regression-testing strategy based on input partitioning and cause–effect graphing of the program specification has been described by Yau and Kishimoto (1987). This strategy derives the input partitions of the modified program using the program specification and code. Then, each new or changed input partition is executed at least once. Symbolic execution is also used to identify the input partitions that are not modified and to aid in test case generation.

A strategy which combines data flow testing with incremental data flow analysis for unit and integration regression testing has been described by Harrold and Soffa (1988). After modifying a module, the effects of the modification on the results of the initial data flow analysis are determined in order to select new and initial definition-use pairs for retesting. Harrold, Gupta and Soffa (1993) have also suggested a methodology for selecting a minimal number of retests that revalidates all requirements of a module. On the other hand, Rothermel and Harrold (1993) have proposed a safe algorithm that uses a module’s dependence graph and selects retests that will cause the modified module to produce different output than the original module.

Path expressions are used by Benedusi, Cimitile and DeCarlini (1988) to represent the program before and after modifications. Then, the two expressions are compared using algebraic operations to determine the paths affected by the modifications. This facilitates the selection of the initial test cases needed for retesting. A similar approach based on semantic differencing has been proposed by Binkley (1992). In this technique, programs are represented by dependence graphs, and program slicing (Gallagher and Lyle, 1991) is used to partition the new program into preserved points and affected points. The test cases selected for retesting are those that test the behaviour of the affected points. These test cases can be run on a smaller program produced by semantic differencing, which captures the behaviour of the affected points only.

Gupta, Harrold and Soffa’s algorithm (1992) uses slicing to detect explicitly definition-use pairs that are directly or indirectly affected by a program modification. The program slice associated with the modified statement/segment is determined by backward and forward walk procedures. The selected test cases will be all those that traverse the slice. Agrawal, Horgan and Krauser (1993) proposed an incremental algorithm which selects test cases whose outputs may be affected by the modifications made to the program.

An approach different from the above-mentioned approaches has been proposed by Fischer

Copyright © 1999 John Wiley & Sons, Ltd. 

(1977) and Fischer, Raji and Chruscicki (1981), and later extended by Hartmann and Robson (1989). It is based on a 0–1 integer programming formulation of the optimal retesting problem, where its solution is the minimum-cardinality subset of test cases that cover all the program segments traced by the initial tests. The integer programming model is based on the control flow graph of the program and the initial set of test cases. Standard linear programming techniques used to solve this model usually terminate with a non-integer solution. To arrive at a 0–1 solution, additional constraints and iterations are needed. Cutting plane methods sometimes work well and sometimes not at all. Moreover, branch and bound algorithms (Horowitz and Sahni, 1987, pp. 370–415) have exponential complexity that makes their execution times unrealistic for large programs.

In this paper, we present two optimization algorithms that are derived from natural and physical phenomena and are adapted to solve the problem of software retesting. The algorithms are a simulated annealing algorithm (SA) and a hybrid genetic algorithm (GA), which are augmented with procedures to guarantee feasible solutions. These algorithms solve the integer programming model of the retesting problem and produce optimal or near-optimal solutions without suffering from intractability. We have applied these algorithms to a variety of program modules and compared them with three well-known methods (branch and bound, slicing, and incremental algorithms). The experimental results show that SA and GA select a low number of test cases (to be rerun) in a reasonable execution time.

This paper is organized as follows. Section 2 describes the integer programming formulation of the optimal retesting problem and its assumptions. Sections 3 and 4 present the simulated annealing and genetic algorithms for solving the problem. Section 5 presents the experimental results. Section 6 contains the conclusions.

## 2. INTEGER PROGRAMMING MODEL OF RETESTING

The integer programming model (Fischer, Raji and Chruscicki, 1981) assumes that the module under consideration is represented by a control flow graph with $M$ program segments, where a program segment represents either a control statement or a contiguous sequence of non-control statements. Also, it is assumed that the set of $N$ test cases used in the initial development of the module have been saved, and that a table of the test cases and the program segments they cover can be determined. Further, the control flow graph enables us to derive information about the reachability of program segments from other segments.

Given that program segment $k$ has been modified, the optimal retesting problem consists of finding values for $(X_1, X_2, \ldots, X_N)$ that minimize the cost function

$$Z = c_1 X_1 + c_2 X_2 + \cdots + c_N X_N \tag{1}$$

subject to the constraints

$$\sum_{j=1}^{N} \mathbf{a}_{ij} X_j \geq b_i; \quad i = 1, \ldots, M \tag{2}$$

where: $X_j = 1$ (or 0) indicates the inclusion (or exclusion) of test case $j$ in the selected subset of retests; $c_j$ is the cost associated with the test case $j$ and will henceforth be assumed to be unity; the matrix $[\mathbf{a}_{ij}]$ is derived directly from the test-segment coverage table, i.e., $\mathbf{a}_{ij} = 1$ if segment $i$ is

Copyright © 1999 John Wiley & Sons, Ltd. 

covered by test case $j$; $b_i = 1$ (or 0) indicates whether segment $i$ needs to (or need not) be covered by the subset of retests due to the modification of segment $k$, where the values $b_i$ are derived from the segment reachability information.

## 3. SIMULATED ANNEALING ALGORITHM

### 3.1. Background

In this subsection, we give background information on the simulated annealing paradigm. In the subsequent subsections, we describe how we adapted this paradigm to solve the optimal retesting problem.

Simulated annealing is based on ideas from statistical mechanics and is motivated by an analogy to the physical annealing of a solid (Kirkpatrick, Gelatt and Vecchi, 1983). To coerce some material into a low-energy state, the material is heated to a high temperature (high-energy state) and then cooled very slowly, allowing it to come to thermal equilibrium at each temperature. At freezing temperatures, the system is expected to acquire a low-energy state.

The behaviour of the system at each fixed temperature in the cooling schedule can be simulated by the Metropolis algorithm (a Monte Carlo algorithm). It starts with a randomly generated (candidate) solution at a high (artificial) temperature, $T(0)$, and then reduces the temperature gradually to a freezing point, $T_f$. At each temperature, $T(i)$, regions in the solution space are searched by the Metropolis algorithm. An iteration of the Metropolis algorithm starts with proposing a random perturbation to the candidate solution and evaluating the resultant change in the energy system. If the change is negative, corresponding to a downhill move in the energy landscape, the perturbation is accepted and the new lower energy configuration becomes the starting point for the next perturbation. Zero change is also accepted. If the energy change is positive, corresponding to an uphill move, the proposed perturbation may be accepted with a temperature-dependent probability. The main advantage of this Metropolis algorithm is that controlled uphill moves can prevent the system from being prematurely trapped in a local minimum-energy state.

Metropolis iterations are repeated many times at each temperature, until equilibrium. The temperature is lowered in each cooling step by using a chosen schedule. After many cooling steps, a freezing point is reached where perturbations do not improve the solution and are no longer accepted. At this point, the solution is expected to have reached a minimum energy value.

Simulated annealing has been adapted to solve many engineering, scientific and operations research problems. Some examples of such applications can be found in Vidal (1993) and in Ebeling *et al.* (1996). In the following subsections, we describe how we adapted simulated annealing (SA) to solve the optimal retesting problem. That is, we present the components of the SA algorithm and the design decisions made with the aim of minimizing the cost function $Z$ (expression (1)). Figure 1 gives an outline of the SA algorithm.

### 3.2. Solution representation and energy function

The system dealt with in the optimal retesting problem is represented by the configuration $X_{\text{soln}} = (X_1, X_2, \ldots, X_N)$ and its energy is given by the cost function $Z$ (expression (1)). A particular

Copyright © 1999 John Wiley & Sons, Ltd. 

```
Initial configuration = random X_soln;
Determine initial temperature T(0);
Determine freezing temperature T_f ;
while (T(i) > T_f  and not converged) do
        repeat
                perturb ( X_soln );
                if (perturbed X_soln  is feasible) then
                        accept_or_not ( )
                else
                        reject_perturbation;
        until equilibrium
        save_bestsofar( );
        check_convergence( );
        T(i+1) = α T(i);  /*cooling schedule*/
Endwhile

procedure  accept_or_not( )
        if (ΔZ ≤ 0) then
                update (X_soln);  /*accept perturbation*/
        else
                if (random(0,1) < e^(-ΔZ/T(i)) ) then
                        update (X_soln)
                else
                        reject_perturbation;
        endif
```

*Figure 1. Summary statement of the simulated annealing algorithm for optimal retesting*

instance of $X_{\text{soln}}$ thus yields a subset of selected test cases, each with index $j$ $(1 \leq j \leq N)$ that corresponds to $X_j = 1$. In the annealing process, $X_{\text{soln}}$ goes through many changes until it reaches the freezing temperature where the final 'optimal' configuration represents the desired solution. However, the initial configuration is generated randomly by assigning each $X_j$ to be 1 or 0 with equal probability.

## 3.3. The Metropolis step and feasibility

The Metropolis step consists of a perturbation operation, an accept/reject criterion and a thermal equilibrium criterion.

Perturbation to $X_{\text{soln}}$ takes place by randomly selecting an index $j$, in the range 1 to $N$, and attempting to include (or exclude) the corresponding test if this test case is currently excluded (or included). That is, we change $X_j$ from $X_j^{\text{old}}$ to the opposite value $X_j^{\text{new}}$. The acceptance criterion outlined in the `accept_or_not()` procedure in Figure 1, checks the change in the cost function, $\Delta Z = c_j(X_j^{\text{new}} - X_j^{\text{old}})$, due to a perturbation. If the change decreases the cost function, the perturbation is accepted and $X_j$ becomes the $X_j^{\text{new}}$. However, if the perturbation causes the cost function to increase, it is accepted only with a probability $e^{-\Delta Z/T(i)}$. Note that for lower temperature values $T(i)$, the probability of accepting uphill moves becomes smaller; at very low (freezing) temperatures, uphill moves (i.e., including more test cases) are no longer accepted.

Copyright © 1999 John Wiley & Sons, Ltd.

Clearly, the classic probabilistic acceptance criterion we have just described would violate the constraints in expression (2) if too many $X_j$ values are changed to 0, leading to an infeasible solution. This is why feasibility is enforced by applying the acceptance criterion only to perturbations that keep $X_{\text{soln}}$ in the feasible region of the search space.

The perturbation-acceptance step is repeated many times at every temperature. Thermal equilibrium is considered reached when $K_{\text{max-att}}$ perturbations are attempted or $K_{\text{max-acc}}$ perturbations are accepted. In this work, we choose these values to be small multiples of $N$; specifically $K_{\text{max-att}} = 5N$ and $K_{\text{max-acc}} = 4N$.

### 3.4. Cooling schedule

The initial temperature $T(0)$ is the temperature that yields a high initial acceptance probability of 0.85 for uphill moves. The freezing point is the temperature at which such a probability is very small ($2^{-30}$), making uphill moves impossible and allowing only downhill moves. The cooling schedule used in this work is simple: $T(i+1) = \alpha T(i)$, with $\alpha = 0.95$.

### 3.5. Convergence

As the annealing algorithm searches the solution space, the best-so-far solution (with the smallest $Z$) found is always saved. This guarantees that the output of the algorithm is the best solution it finds regardless of the temperature it terminates at. Convergence is then detected when the algorithm does not improve on the best-so-far solution for a number of temperatures, say 20, in the colder part of the annealing schedule.

## 4. HYBRID GENETIC ALGORITHM

### 4.1. Background

In this subsection, we give background information on the genetic algorithm paradigm. In the subsequent subsections, we describe how we adapted this paradigm to solve the optimal retesting problem.

Genetic algorithms are based on the mechanics of natural evolution (Goldberg, 1989). They mimic natural populations' reproduction and selection operations to achieve efficient and robust optimization. Through their artificial evolution, successive generations search for beneficial adaptations in order to solve a problem. Each generation consists of a population of chromosomes, also called individuals, and each chromosome represents a possible solution to the problem. The initial generation consists of randomly-created individuals. Each individual acquires a fitness level that is usually based on a cost function given by the problem under consideration.

The Darwinian principle of reproduction and survival of the fittest and the genetic operations of recombination (crossover) and mutation are used to create a new offspring population from the current population. The reproduction operation involves selecting, in proportion to fitness, a chromosome from the current population of chromosomes, and allowing it to survive by copying it into the new population. Then, two mates are randomly selected from this population, and

Copyright © 1999 John Wiley & Sons, Ltd. 

```
Random generation of initial population, size POP;
Evaluate fitness of individuals;
repeat
    Rank individuals and allocate reproduction trials;
    for (i=1 to POP step 2) do
        Randomly select 2 parents from the list of reproduction trials;
        Apply crossover and mutation;
    endfor
    Evaluate fitness of offspring;
    Check feasibility of individuals;
    Do feasibilization, penalization, and hill-climbing;
    Preserve the fittest-so-far (elitism);
until (convergence criterion is satisfied)
Solution = Fittest.
```

*Figure 2. Summary statement of the hybrid genetic algorithm for optimal retesting*

crossover and mutation are carried out to create two new offspring chromosomes. Crossover involves swapping two randomly located sub-chromosomes (within the same boundaries) of the two mating chromosomes. Mutation is applied to randomly-selected genes, where the values associated with such genes are randomly changed to other values within an allowed range. The offspring population replaces the parent population, and the process is repeated for many generations with the aim of maximizing the fitness of the individuals.

Genetic algorithms have been adapted for solving a variety of engineering, science and operational research problems. Some examples of such applications can be found in Davis (1991), Ebeling *et al.* (1996), Fogel (1998) and Baeck (1997). In the following subsections, we present the components of a hybrid genetic algorithm (GA) and the design decision made for minimizing the cost function $Z$ (expression (1)). The genetic algorithm is hybridized with a procedure that accounts for the likelihood of producing infeasible individuals after crossover and mutation. Figure 2 outlines this hybrid GA.

## 4.2. Population, chromosomal representation and fitness

The population consists of arrays of possible test cases. The cardinality of the population is represented by POP, and a test case is regarded as being a gene in the population. A GA population is represented as an array of POP individuals. Each individual in the population is encoded as an $N$-element (gene) array $(X_1, X_2, \ldots, X_N)$ that corresponds to a candidate solution for the optimal retesting problem. An element (gene) $X_j = 1$ (or 0) indicates the inclusion (or exclusion) of test case $j$ in (or from) the selected subset of retests. The initial population of individuals is randomly generated.

We use $1/Z$ as the fitness value of an individual, where $Z$ is given by expression (1). We attempt to maximize the fitness value.

Copyright © 1999 John Wiley & Sons, Ltd. 

### 4.3. Reproduction scheme

The whole population is considered to be a single reproduction unit within which random selection is performed. Our reproduction scheme involves elitist ranking, followed by random selection of mates from the list of reproduction trials (or copies) assigned to the ranked individuals. In the ranking scheme (Baker, 1985), the individuals are sorted by fitness values. After sorting, each individual is assigned a rank based on a scale of equidistant values for the population. The ranks assigned to the fittest and least-fit individuals are 1.2 and 0.8, respectively. Individuals with ranks greater than 1 are first assigned single copies. Then, the fractional part of their ranks and the ranks of the lower half of individuals are treated as probabilities for random assignment of copies.

Elitism is used to exploit good building blocks and to ensure that good candidate solutions are preserved. This is done by replacing the least-fit individual with the fittest-so-far individual if the latter is better than the current-fittest.

### 4.4. Genetic operators

The genetic operators employed in GA are crossover and mutation at the rates 0.7 and 0.02 (Grefenstette, 1986), respectively. The application of the operators starts with randomly selecting pairs of individuals from the mating pool. Each pair of these chromosomes undergoes crossover, where an integer position $k$ along the chromosome is selected at random between 1 and $N$, and all genes between $k+1$ and $N$ are swapped to create two new chromosomes. Then, standard mutation is applied, where a randomly selected gene (test case), $X_j$, is flipped to the opposite value (changed from included to excluded or the other way round).

### 4.5. Feasibilizing, penalizing and hill climbing

The genetic operators can produce infeasible solutions. We dealt with this problem by using a technique that combines procedures for feasibilizing and penalizing. This technique which we proposed previously (Easton and Mansour, 1993) is illustrated in Figure 3.

At the detection of an infeasible candidate solution (chromosome), the GA algorithm computes a simple penalty based on the maximum shortage difference,

$$\left( \sum_{j=1}^{N} \mathbf{a}_{ij} X_j - b_i \right) \tag{3}$$

which represents the minimum number of test cases needed by this candidate solution to make it satisfy all the constraints in expression (2). This difference is added to $Z$. The infeasible chromosome is then allowed to enter the population intact. In this way the penalized infeasible chromosome will have a lower probability of survival. However, 50% of the infeasible offspring are randomly selected for heuristic feasibilization.

A random feasibilization heuristic is applied by randomly selecting a violated (infeasible) row from expression (2), and then by randomly selecting a gene capable of reducing this violation. That is, it increases by 1 the number of test cases to be included in the retest subset. The process is then repeated until all constraints are satisfied. This helps to ensure that the population includes some

Copyright © 1999 John Wiley & Sons, Ltd. 

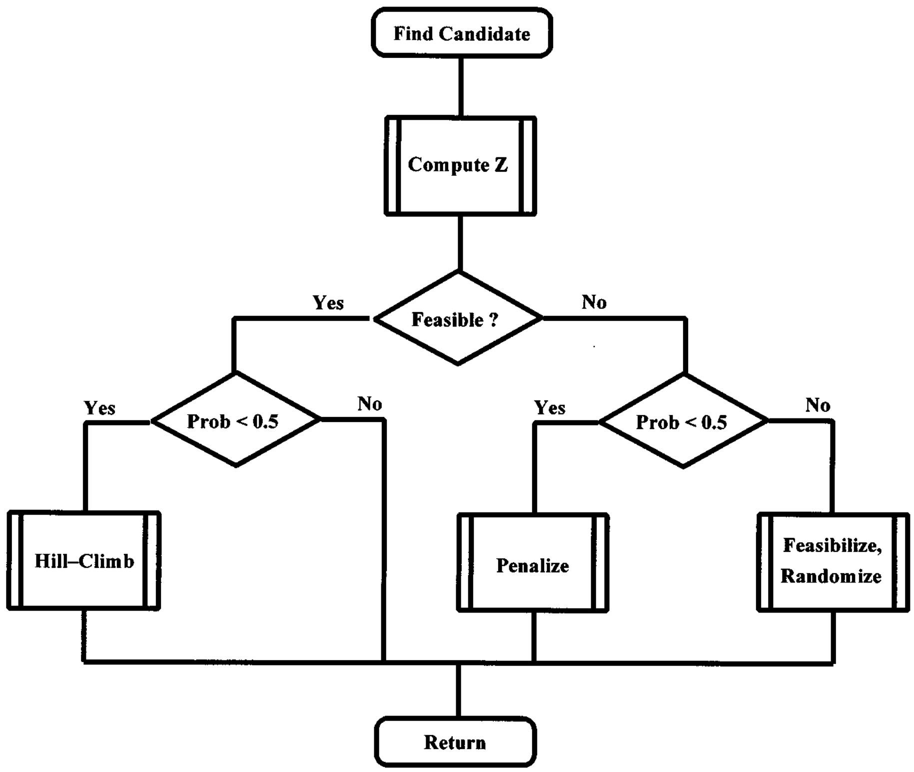


*Figure 3. Flow chart of feasibilizing, penalizing and hill climbing*

feasible candidates that could be exploited in subsequent generations, and helps maintain the locus of the search near the feasible region.

To refine solution quality, we incorporated a fast problem-specific hill-climbing procedure that can improve an individual's fitness value. The GA randomly selects one half of all feasible solutions formed in each generation and applies the following procedure that searches nearby solution space using a simple 'drop one' methodology. One by one, the value of each gene is provisionally decremented. That is, a test case is provisionally omitted. If the resulting solution is feasible, the provisional change is made permanent and then a new solution becomes the incumbent. If infeasible, the procedure restores the gene to its original value and selects another gene. The procedure terminates when no single test case (gene) in the current candidate solution can be omitted (decremented) to form a new, higher-fitness and feasible solution. This hill-climbing procedure enables individuals to climb the peaks rapidly which speeds up the evolution process.

## 4.6. Convergence

Convergence is detected when the best-so-far candidate solution does not change its $Z$ value for 20 generations.

Copyright © 1999 John Wiley & Sons, Ltd.

## 5. EXPERIMENTAL RESULTS

### 5.1. Situation

In this section we experimentally compare the results of SA and GA with those of a branch and bound (BB) algorithm used by Fischer, Raji and Chruscicki (1981) for small-size modules. For larger-size modules, the BB algorithm fails to converge in a reasonable time due to exponential explosion. Also, we compare our algorithms with well-known approaches based on slicing and incremental algorithms (Gupta, Harrold and Soffa, 1992; Agrawal, Horgan and Krauser, 1993).

The 31 test modules used were arbitrarily designed. Further, we assumed that the test case segment coverage of the initial test cases satisfies at least an all-statements coverage criterion. More test cases were added in an arbitrary way to the basic all-statements coverage for the majority of the 31 modules. The objective was to provide more diversity in these test problems for the retesting algorithms. In all experiments, the following were assumed given for each module: a control flow graph of $M$ segments, the number of the modified segment ($S_{\mathrm{mod}}$), a table of $N$ initial test cases, and the initial test segment coverage information. In all the figures given below, a module is characterized by a name $m_i$ and the pair $(M, N)$.

### 5.2. Comparison with the BB algorithm

In the following experiments, all three algorithms (BB, SA and GA) find optimal numbers of test cases. Consequently, the comparison reports on performance only. The experiments were done on an AViion 5000 UNIX machine.

Figures 4 and 5 give the execution times of the three algorithms for seven small-size modules. These results show that BB would be the fastest algorithm for small-size modules and small number of test cases. But, as the module size and the number of test cases grow, GA clearly becomes the fastest. SA exhibits a similar behaviour to that of GA, but it is slower.

Figure 6 gives the execution times of SA and GA for four medium-size modules whose control flow graphs were randomly generated. Again, GA is faster than SA for all four cases and the execution times for BB are not included because they are unreasonably large.

### 5.3. Comparison with slicing and incremental algorithms

The experiments comparing GA and SA with the well-known slicing and incremental algorithms were done on a PC running an INTEL 486 100 MHz CPU. Twenty different program modules, including small sizes ($m_{12}$–$m_{25}$) and medium sizes ($m_{26}$–$m_{31}$), were used.

Table 1 gives the execution times, in seconds, and the number of selected retests for the four algorithms on the 20 modules. A typical sample of these results is illustrated in Figures 7 and 8. These results show that for most modules, the slicing algorithm selects high numbers of retests. The incremental algorithm yields better numbers of retests for similar execution times. The genetic and simulated annealing algorithms select the least number of retests for the great majority of the modules, although they have the tendency to be slower.

Copyright © 1999 John Wiley & Sons, Ltd. 

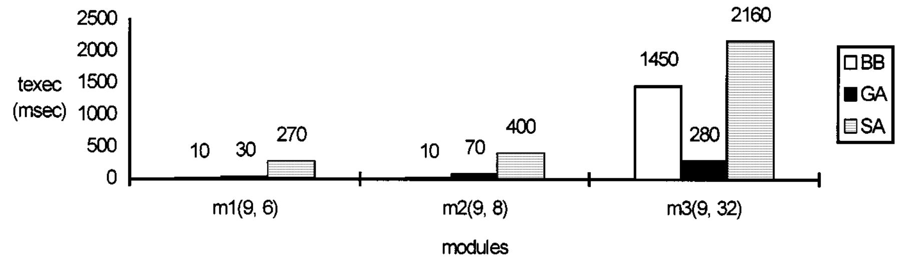


*Figure 4. Execution times for algorithms BB, GA and SA for modules* $m_1$ *to* $m_3$

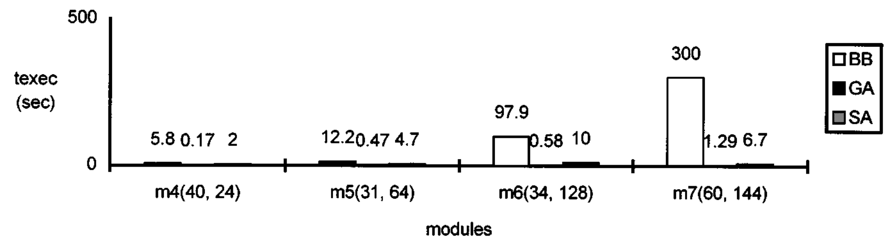


*Figure 5. Execution times for algorithms BB, GA and SA for modules* $m_4$ *to* $m_7$

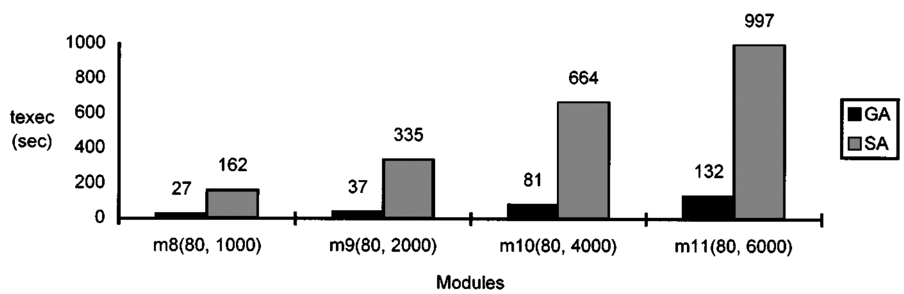


*Figure 6. Execution times for algorithms GA and SA for modules* $m_8$ *to* $m_{11}$

## 5.4. Discussion of the results

Clearly, the GA and SA algorithms take a minimization approach to selecting retests. They select only the test cases necessary to execute (once) the modified segment, $S_{mod}$, and all the segments reachable from $S_{mod}$ in the control flow graph. This behaviour is similar to that of the BB algorithm that suffers from exponential explosion for large problems. But, it is different from the behaviour of the slicing and incremental algorithms that select test cases based on how they affect the execution slices and the output of the module. That is, if $S_{mod}$ involves dense data dependencies with the segments that precede it and those that follow it in the execution paths, the number of retests selected

Table 1. The execution times ($t_{exec}$, in seconds) and the number of selected retests (#$R$) for the four algorithms

| | | | | Slicing | | Incremental | | Genetic | | Annealing | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | $M$ | $N$ | $S_{mod}$ | $t_{exec}$ | #$R$ | $t_{exec}$ | #$R$ | $t_{exec}$ | #$R$ | $t_{exec}$ | #$R$ |
| $m_{12}$ | 8 | 6 | 3 | 0.3 | 5 | 0.3 | 4 | 8.0 | 2 | 0.3 | 2 |
| $m_{13}$ | 8 | 32 | 8 | 0.8 | 26 | 0.8 | 8 | 6.7 | 1 | 2.1 | 1 |
| $m_{14}$ | 10 | 15 | 8 | 0.3 | 5 | 0.6 | 5 | 4.9 | 1 | 1.0 | 1 |
| $m_{15}$ | 10 | 27 | 10 | 0.5 | 8 | 1.0 | 8 | 1.4 | 1 | 1.8 | 1 |
| $m_{16}$ | 14 | 24 | 13 | 1.4 | 6 | 1.4 | 6 | 4.2 | 2 | 1.7 | 1 |
| $m_{17}$ | 14 | 32 | 14 | 1.5 | 32 | 0.7 | 8 | 4.0 | 1 | 2.3 | 1 |
| $m_{18}$ | 21 | 9 | 12 | 1.3 | 5 | 1.3 | 5 | 3.5 | 2 | 0.5 | 2 |
| $m_{19}$ | 21 | 18 | 19 | 2.0 | 11 | 2.0 | 7 | 7.4 | 1 | 1.3 | 1 |
| $m_{20}$ | 24 | 18 | 9 | 0.9 | 5 | 0.9 | 3 | 9.2 | 3 | 1.1 | 3 |
| $m_{21}$ | 24 | 23 | 12 | 1.3 | 5 | 1.3 | 3 | 9.8 | 2 | 2.0 | 3 |
| $m_{22}$ | 34 | 58 | 29 | 5.6 | 12 | 3.1 | 8 | 9.9 | 2 | 4.3 | 2 |
| $m_{23}$ | 34 | 126 | 32 | 7.0 | 29 | 5.2 | 8 | 9.1 | 1 | 10.6 | 1 |
| $m_{24}$ | 40 | 20 | 21 | 2.3 | 7 | 2.3 | 3 | 4.6 | 3 | 1.4 | 3 |
| $m_{25}$ | 40 | 28 | 39 | 4.6 | 9 | 4.2 | 3 | 5.3 | 1 | 2.0 | 1 |
| $m_{26}$ | 45 | 32 | 27 | 2.9 | 4 | 2.9 | 4 | 7.3 | 4 | 2.5 | 3 |
| $m_{27}$ | 45 | 96 | 43 | 7.0 | 16 | 7.0 | 8 | 6.3 | 4 | 8.2 | 3 |
| $m_{28}$ | 56 | 34 | 41 | 5.5 | 8 | 6.7 | 9 | 5.7 | 3 | 2.8 | 3 |
| $m_{29}$ | 56 | 108 | 50 | 9.4 | 22 | 8.2 | 5 | 5.9 | 2 | 9.6 | 2 |
| $m_{30}$ | 60 | 90 | 30 | 4.9 | 15 | 4.9 | 3 | 7.9 | 4 | 7.6 | 3 |
| $m_{31}$ | 60 | 145 | 46 | 9.2 | 22 | 6.5 | 2 | 6.4 | 3 | 12.9 | 3 |

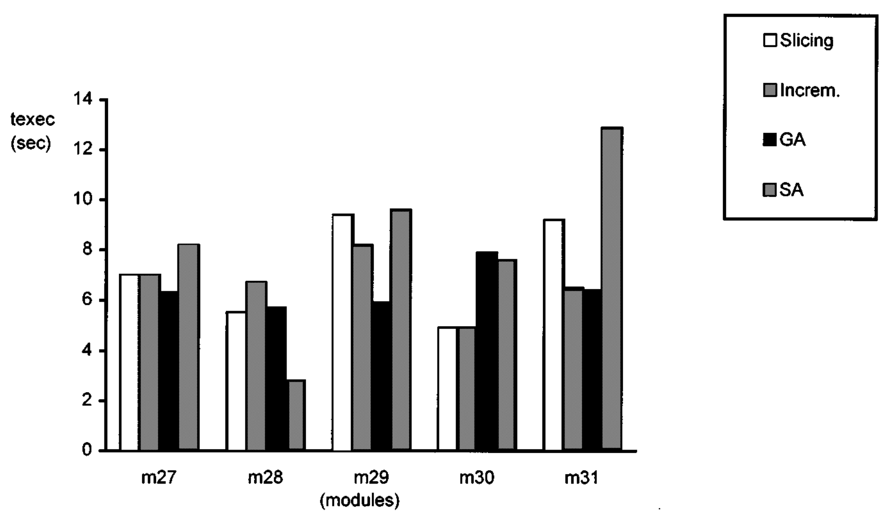


*Figure 7. The execution times for the four algorithms for modules* $m_{27}$ *to* $m_{31}$

Copyright © 1999 John Wiley & Sons, Ltd.

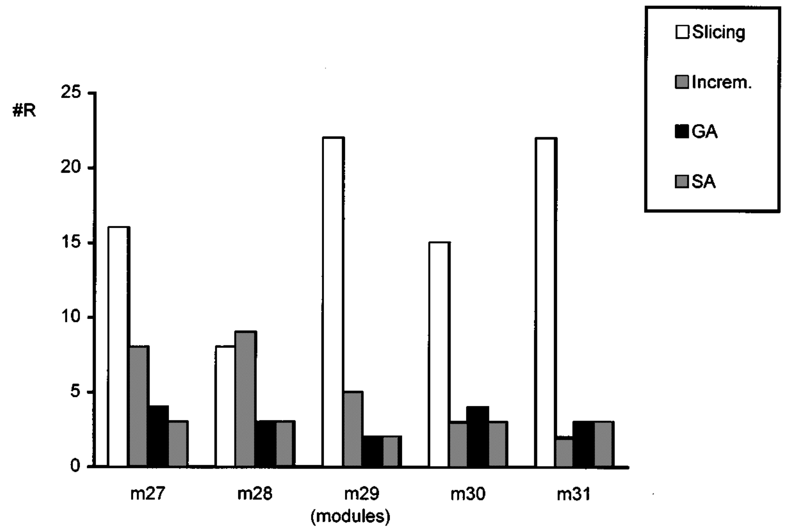


*Figure 8. The number of selected retests for the four algorithms for modules* $m_{27}$ *to* $m_{31}$

by the slicing and incremental algorithms are likely to increase. In contrast, GA and SA are based only on control dependencies and would still aim for selecting the minimal number of retests. This behaviour points to an advantage and a disadvantage. The advantage is that GA and SA should be particularly useful for problems that are large in terms of the number of segments and the number of initial test cases. They should also be useful for problems that require low regression testing time. The disadvantage is that GA and SA might miss useful retests. The experimental results showed another disadvantage of GA and SA: their tendency to be slower than the slicing and incremental algorithms.

The 31 test modules were designed in a way that offers some variety in the control structure. Inspecting these module structures does not reveal biases in their choice in favour of the GA and SA algorithms. However, we note that when the number of segments reachable from $S_{mod}$ is small, GA and SA are likely to select a small number of retests. Further, we assumed all-statements or all-branches criteria for the initial test cases. But, any other criterion could have been used and would not have changed the fundamental property of GA and SA as minimization algorithms.

The overheads associated with the operation of GA and SA are: the generation of the program's control flow graph; the determination of the segment test case coverage matrix; the determination of the reachability terms, $b_i$ (expression (2)), by a breadth-first search of the control flow graph starting at $S_{mod}$. Obviously, if the program's graph and coverage matrix are saved from the initial development phase, the overhead would be significantly reduced.

## 6. CONCLUSIONS

Simulated annealing and genetic algorithms have been presented for reducing the number of test cases that have to be rerun for regression testing a modified program module. The algorithms are

Copyright © 1999 John Wiley & Sons, Ltd. 

based on an integer programming problem formulation and on a control flow graph representation of a module.

We have applied the SA and GA algorithms to various module sizes and sets of initial test cases and compared their results with those of three well-known algorithms. The experimental results show that SA and GA select optimal or near-optimal numbers of retests and take reasonable execution times. Also, in contrast to mathematical optimization methods, their complexity does not grow exponentially for realistic and large module sizes. This makes them useful for large problems and for those that require minimum regression testing time.

## Acknowledgements

We thank the Lebanese National Council for Scientific Research for their support. We thank Ghinwa Baradhi for implementing the slicing and incremental algorithms. Nashat Mansour thanks Amrit Goel of Syracuse University for supporting the initial steps of the work reported here.

## References

Agrawal, H., Horgan, J. R., and Krauser, E. W. (1993) 'Incremental regression testing', in *Proceedings Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, pp. 348–357.

Baeck, E. (Ed) (1997) *Proceedings of the International Conference on Genetic Algorithms*, Morgan Kaufman Publishers, Inc., San Mateo CA, 800 pp.

Baker, J. E. (1985) 'Adaptive selective methods for genetic algorithms', in *Proceedings of the International Conference on Genetic Algorithms*, Morgan Kaufmann Publishers, Inc., San Mateo CA, pp. 101–111.

Benedusi, P., Cimitile, A. and DeCarlini, U. (1988) 'Post-maintenance testing based on path change analysis', in *Proceedings Conference on Software Maintenance—1988*, IEEE Computer Society Press, Los Alamitos CA, pp. 352–368.

Bennett, K. H. (1991) 'The software maintenance of large software systems: management, methods and tools', *Reliability Engineering and Systems Safety*, **32**(2), 135–154.

Binkley, D. (1992) 'Using semantic differencing to reduce the cost of regression testing', in *Proceedings Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, pp. 41–50.

Chen, Y., Rosenblum, D. S. and Vo, K. (1994) 'TestTube: a system for selective regression testing', in *Proceedings International Conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos CA, pp. 211–220.

Davis, L. (Ed) (1991) *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, New York NY, 385 pp.

Easton, F. and Mansour, N. (1993) 'A distributed genetic algorithm for employee staffing and scheduling problems', in *Proceedings of the International Conference on Genetic Algorithms*, Morgan Kaufmann Publishers, Inc., San Mateo CA, pp. 360–367.

Ebeling, W., Rechenberg, I., Schwefel, H. P. and Voigt, H. M. (Eds) (1996) *Parallel Problem Solving from Nature*, vol. 1141 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1050 pp.

Fischer, K. (1977) 'A test selection method for the validation of software maintenance modifications', in *Proceedings International Computer Software and Applications Conference, COMPSAC 77*, IEEE Computer Society Press, Los Alamitos CA, pp. 421–426.

Fischer, K., Raji, F. and Chruscicki, A. (1981) 'A methodology for re-testing modified software', in *Proceedings National Telecommunications Conference*, IEEE Press, Piscataway NJ, pp. B6.3.1–B6.3.6.

Fogel, D. (Ed) (1998) *Proceedings IEEE International Conference on Evolutionary Computation*, IEEE Press, Piscataway NJ, 845 pp.

Gallagher, K. B. and Lyle, J. R. (1991) 'Using program slicing in software maintenance', *IEEE Transactions on Software Engineering*, **17**(8), 752–761.

Goldberg, D. E. (1989) *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, Reading MA, 412 pp.

Grefenstette, J. J. (1986) 'Optimization of control parameters for genetic algorithms', *IEEE Transactions on Systems, Man, and Cybernetics*, **16**(1), 122–128.

Copyright © 1999 John Wiley & Sons, Ltd. 

Gupta, R., Harrold, M. J. and Soffa, M. L. (1992) ‘An approach to regression testing using slicing’, in *Proceedings Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, pp. 299–308.

Harrold, M. J. and Soffa, M. L. (1988) ‘An incremental approach to unit testing during maintenance’, in *Proceedings Conference on Software Maintenance—1988*, IEEE Computer Society Press, Los Alamitos CA, pp. 362–367.

Harrold, M. J., Gupta, R. and Soffa, M. L. (1993) ‘A methodology for controlling the size of a test suite’, *ACM Transactions on Software Engineering and Methodology*, **2**(3), 270–285.

Hartmann, J. and Robson, D. J. (1989) ‘Revalidation during the software maintenance phase’, in *Proceedings Conference on Software Maintenance—1989*, IEEE Computer Society Press, Los Alamitos CA, pp. 70–79.

Horowitz, E. and Sahni, S. (1987) *Fundamentals of Computer Algorithms*, Computer Science Press, Rockville MA, 626 pp.

Hsia, P., Li X., Kung, D. C., Hsu, C.-T., Li, L., Toyoshima, Y. and Chen, C. (1997) ‘A technique for the selective revalidation of OO software’, *Journal of Software Maintenance*, **9**(4), 217–233.

IEEE (1990) *Glossary of Software Engineering Terminology, IEEE Standard 610.12*, IEEE Press, New York NY, 84 pp.

Kirkpatrick, S., Gelatt, C. D. and Vecchi, M. P. (1983) ‘Optimization by simulated annealing’, *Science*, **220**(4598), 671–680.

Leung, H. K. N. and White, L. J. (1989) ‘Insights into regression testing’, in *Proceedings Conference on Software Maintenance—1989*, IEEE Computer Society Press, Los Alamitos CA, pp. 60–69.

Leung, H. K. N. and White, L. J. (1990) ‘Insights into testing and regression testing global variables’, *Journal of Software Maintenance*, **2**(4), 209–222.

Leung, H. K. N. and White, L. J. (1992) ‘A firewall concept for both control-flow and data-flow in regression integration testing’, in *Proceedings Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, pp. 262–271.

Rothermel, G. and Harrold, M. J. (1993) ‘A safe, efficient algorithm for regression test selection’, in *Proceedings Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, pp. 358–367.

Rothermel, G. and Harrold, M. J. (1996) ‘Analyzing regression test selection techniques’, *IEEE Transactions on Software Engineering*, **22**(8), 529–551.

Vidal, R. (Ed) (1993) *Applied Simulated Annealing*, Springer-Verlag, New York NY, 358 pp.

White, L. J., Narayanswamy, V., Friedman, T., Kirschenbaum, M., Piwowarski, P. and Oha, M. (1993) ‘Test manager: a regression testing tool’, in *Proceedings Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, pp. 338–347.

Yau, S. and Kishimoto, Z. (1987) ‘A method for revalidating modified programs in the maintenance phase’, in *Proceedings International Computer Software and Applications Conference, COMPSAC87*, IEEE Computer Society Press, Los Alamitos CA, pp. 272–277.

**Authors’ biographies:**

**Nashat Mansour** is an Associate Professor of Computer Science at the Lebanese American University, Lebanon. His research interests include software testing and maintenance, and natural computation algorithms. He received his B.E. and M.S. degrees in Electrical Engineering from the University of New South Wales, Australia, and an M.S. in Computer Engineering and Ph.D. in Computer Science from Syracuse University, New York. His email address is: nmansour@lau.edu.lb

Copyright © 1999 John Wiley & Sons, Ltd. 

**Khalid El-Fakih** is a Verification Engineer at Cambrian Systems Corporation, Canada. His research interest is formal software validation methods. He received his B.S. and M.S. degrees in Computer Science from the Lebanese American University in 1991 and 1995, respectively. His email address is: kelfakih@cambriansys.com

Copyright © 1999 John Wiley & Sons, Ltd. 